SEHH2239
Data Structures

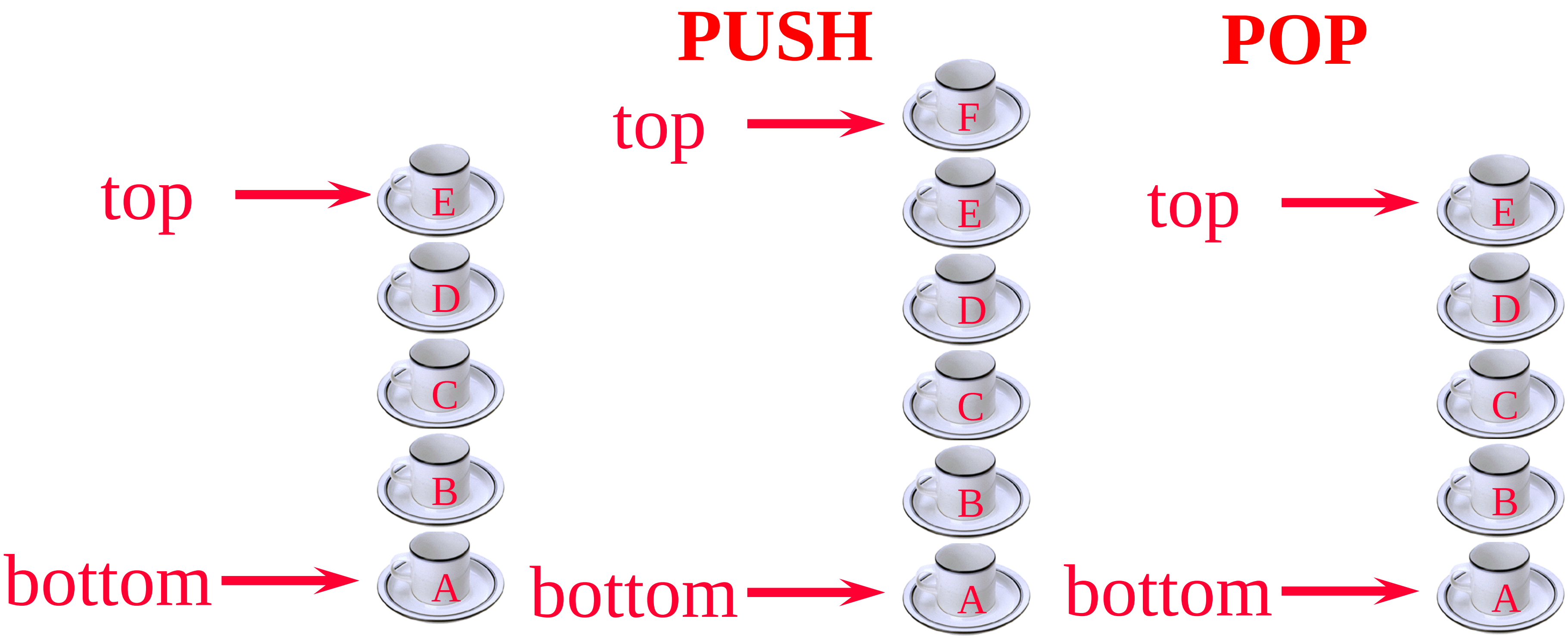
Lecture 7

Stacks and Queue

- Stacks
  - Linear list.
  - One end is called top.
  - Other end is called bottom.
  - **Additions to and removals from the top end only.**
- Queue
  - Linear list.
  - One end is called front.
  - Other end is called rear.
  - **Additions are done at the rear only.**
  - **Removals are made from the front only.**

Stack

Stack Of Cups



- Add a cup to the stack. (F is added) -- **PUSH**
- Remove a cup from the stack. (F is removed) -- **POP**
- A stack is a **LIFO** list. (Last In First Out)

The Stack Abstract Data Type

Idea: If we enforce the
LIFO principle, it becomes a stack.

A stack S is an **abstract data type** (**ADT**) that supports following two fundamental methods:

push(o): Insert object o at the top of the stack
Input: Object; **Output**: None.

pop(): Remove from the stack and return the top object on the stack; an error occurs if the stack is empty.
Input: None; **Output**: Object

The Stack Abstract Data Type

Other supporting methods:

peek() / top(): Return the top object on the stack, without removing it; an error occurs if the stack is empty.

Input: None; Output: Object

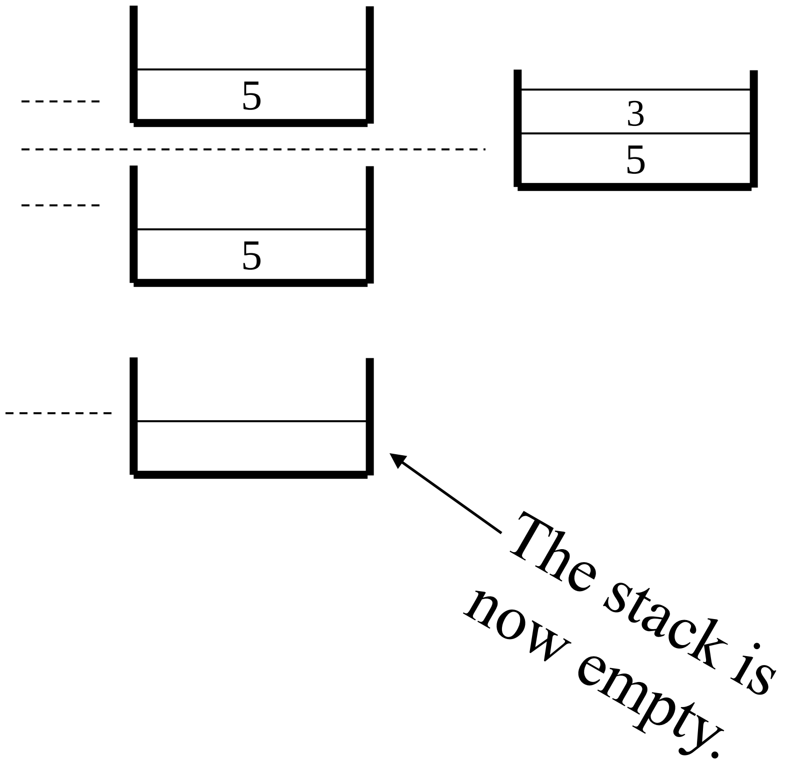
empty() / isEmpty(): Return a Boolean indicating if the stack is empty.
Input: None; Output: Boolean

Optional

size(): Return the number of objects in the stack.
Input: None; Output: Integer

This table shows a series of stack operations and their effects. The stack is initially empty.

Operation	Output	S
push(5)	-	(5)
push(3)	-	(5,3)
pop()	3	(5)
push(7)	-	(5,7)
pop()	7	(5)
top()	5	(5)
pop()	5	()
pop()	“error”	()
isEmpty()	true	()
push(9)	-	(9)
push(7)	-	(9,7)
push(3)	-	(9,7,3)
push(5)	-	(9,7,3,5)
size()	4	(9,7,3,5)
pop()	5	(9,7,3)
push(8)	-	(9,7,3,8)
pop()	8	(9,7,3)
pop()	3	(9,7)



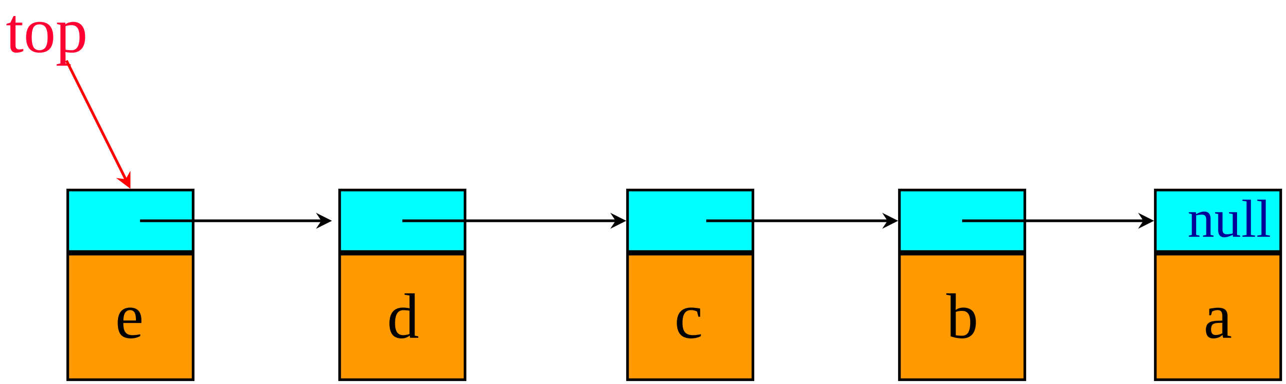
Linked Chain Implementation of Stack

Linked Implementation

- Use the Node class to create an element in stack.
- Use a pointer top to point at the top element.
  - Stack elements are in Node objects.
  - Top element is in top.element.
  - Bottom element is in top.next… structure

    e.g. if there are five elements in stack, size is 5, the bottom element is represented by top.next.next.next.next.

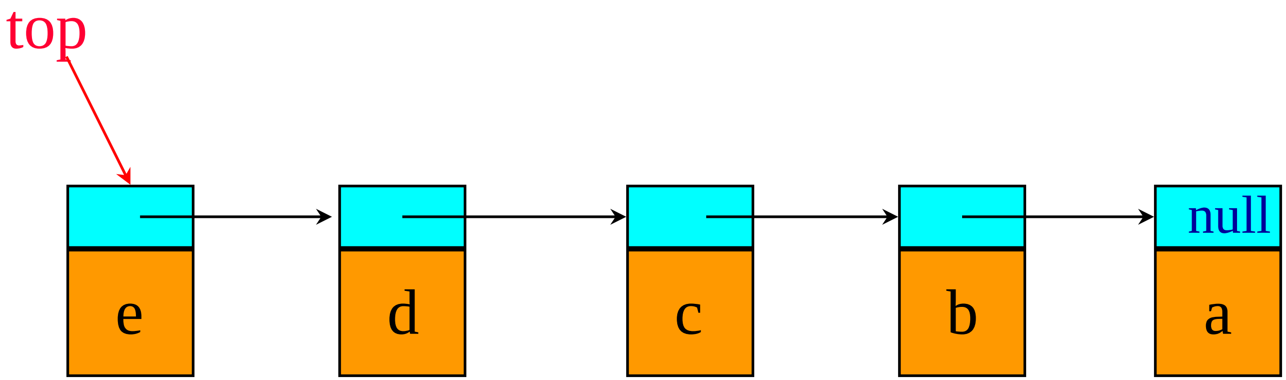
Linked Implementation




```
class Node:
    def __init__(self, el = None, n = None):
        self.next = n
        self.element = el

class LinkedStack:
    def __init__(self):
        self.top = None
        self.size = 0
```

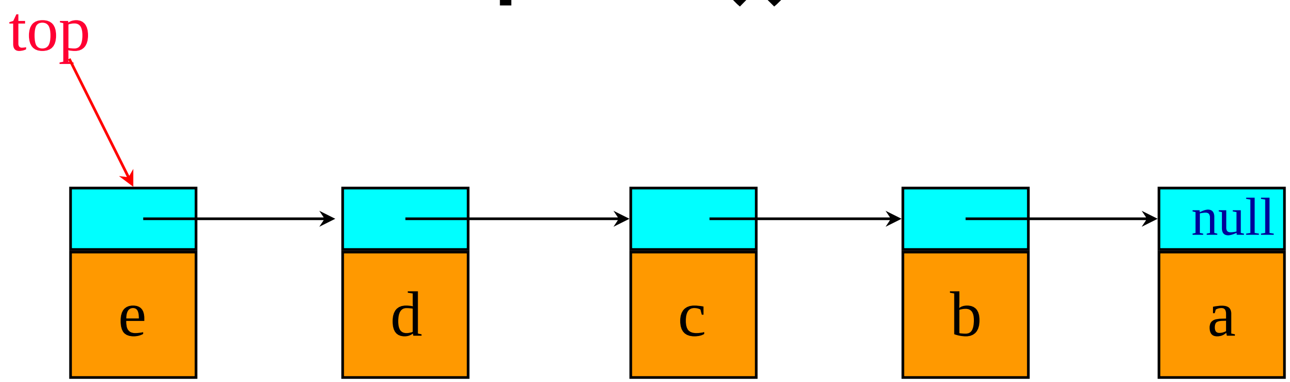
LinkedStack.py

push(…)



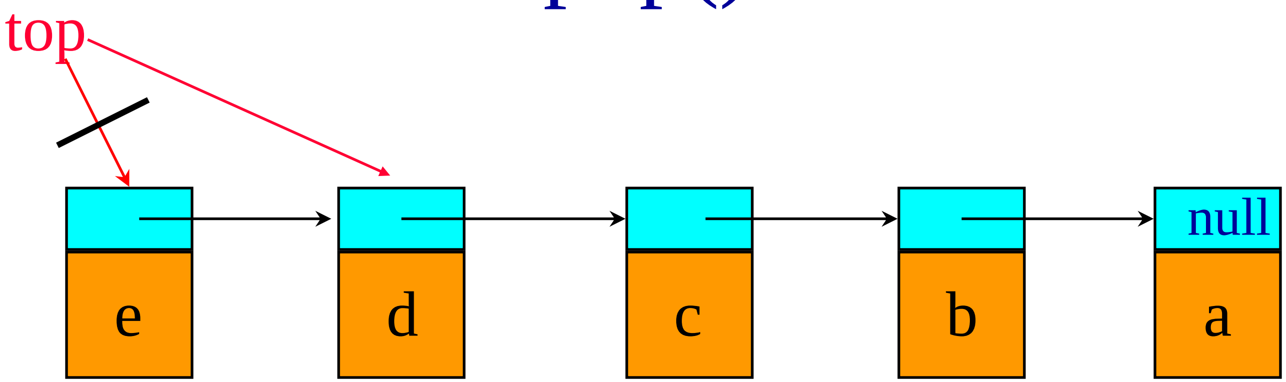

```
#add theElement to the top of the stack
def push(self, element):
     newNode = Node(element)
     newNode.next = self.top
     self.top = newNode
     self.size += 1
```

peek()




```
#return top element of stack
def peek(self):
      if (self.empty()):
         return ("Empty Stack")
      else:
          return self.top.element;
```

pop()




```
#remove top element of stack and return it
def pop(self):
      if (self.empty()):
            return ("Empty Stack")
      topelement = self.top.element
      self.top = self.top.next
      self.size -= 1
      return topelement
```

empty()

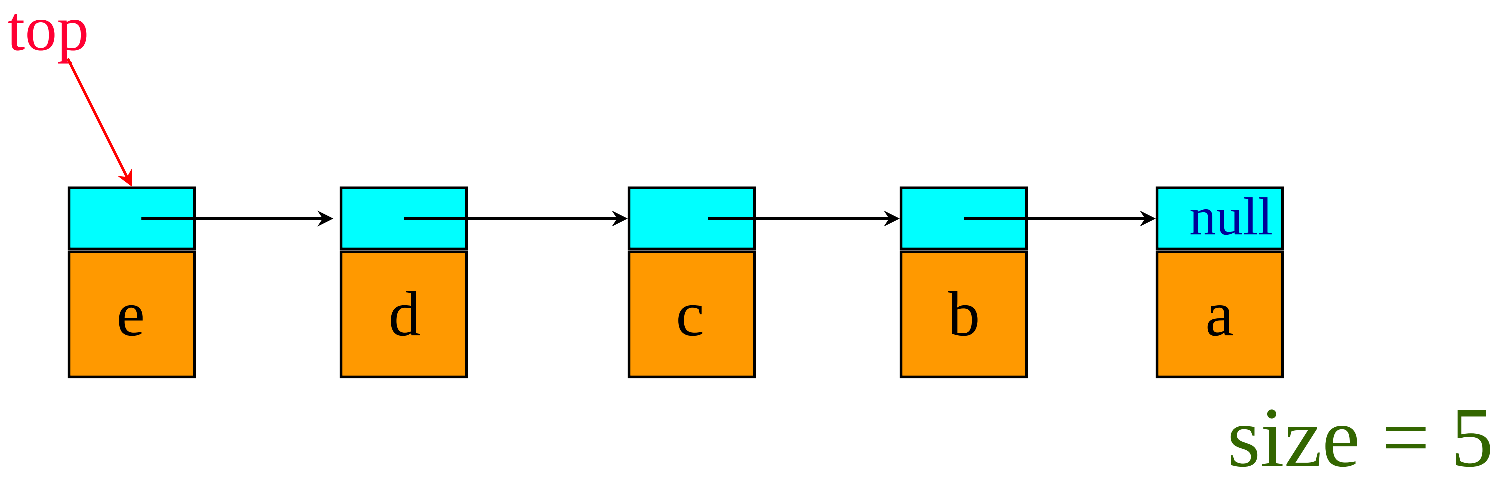
top

null

size = 0

```
#return true if the stack is empty
def empty(self):
   return self.size == 0
```

stack_size()

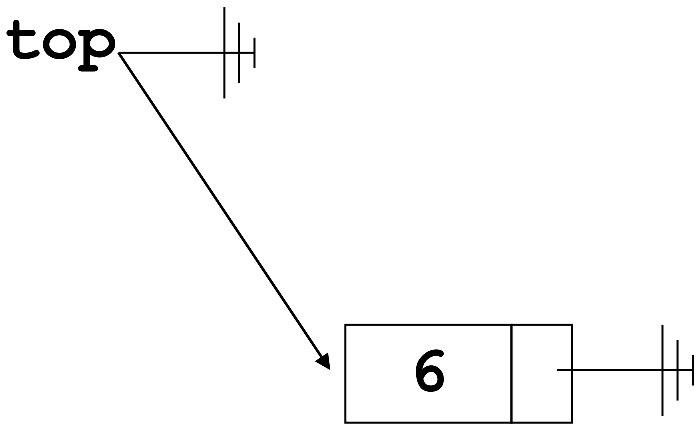



```
#return the size of the stack
def stack_size(self):
      return self.size
```

Linked Stack Example

Python Code

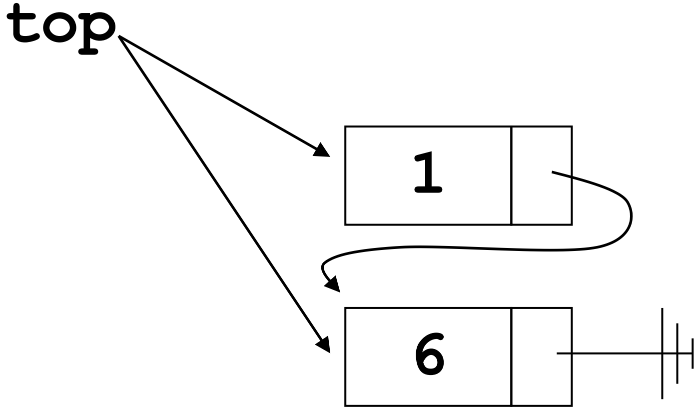
```
st = LinkedStack()
st.push(6);
```



Linked Stack Example

Python Code

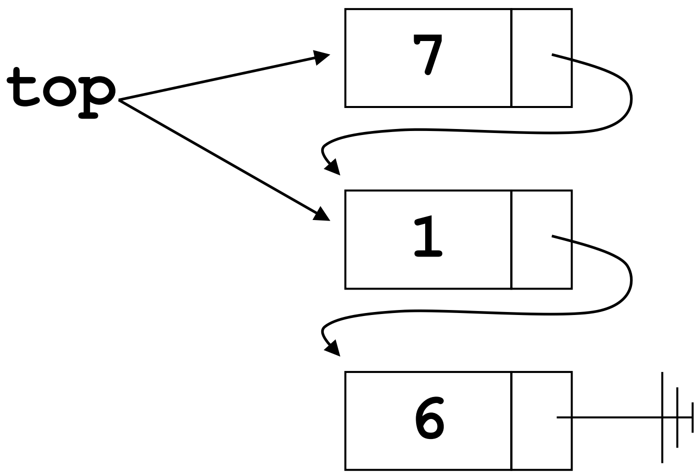
```
st = LinkedStack()
st.push(6);
st.push(1);
```



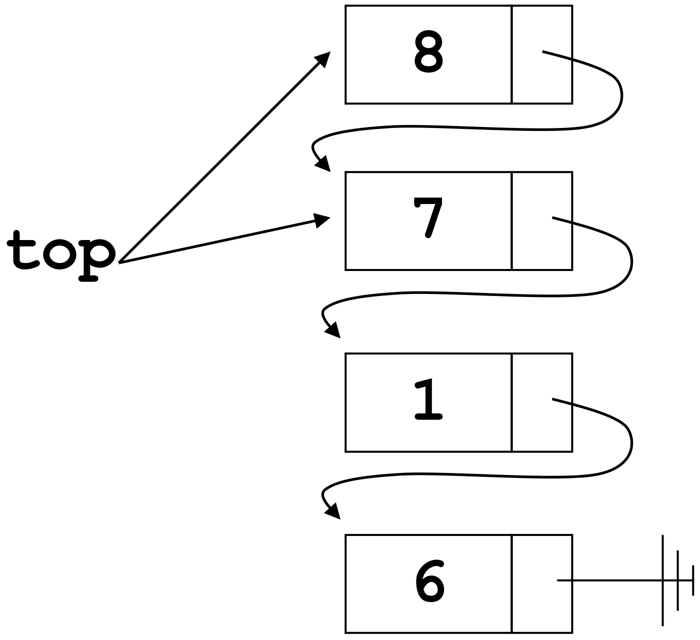
Linked Stack Example

Python Code

```
st = LinkedStack()
st.push(6);
st.push(1);
st.push(7);
```



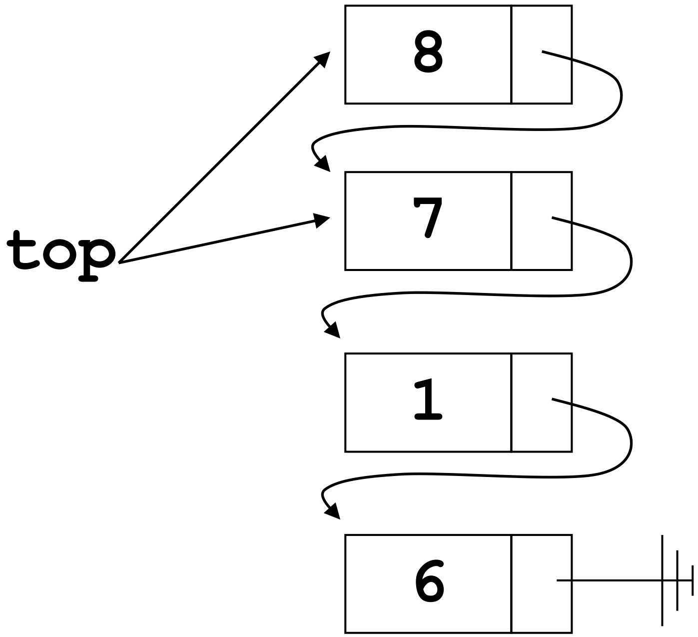
Linked Stack Example



Python Code

```
st = LinkedStack()
st.push(6);
st.push(1);
st.push(7);
st.push(8);
```

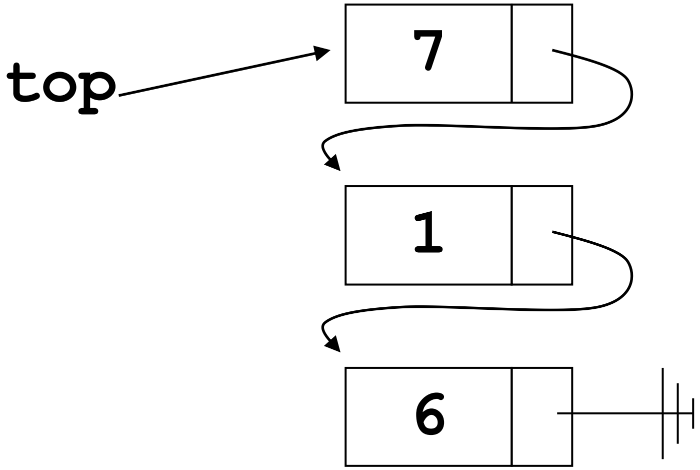
Linked Stack Example



Python Code

```
st = LinkedStack()
st.push(6);
st.push(1);
st.push(7);
st.push(8);
st.pop();
```

Linked Stack Example



Python Code

```
st = LinkedStack()
st.push(6);
st.push(1);
st.push(7);
st.push(8);
st.pop();
```

Array Implementation of Stack

Array Implementation

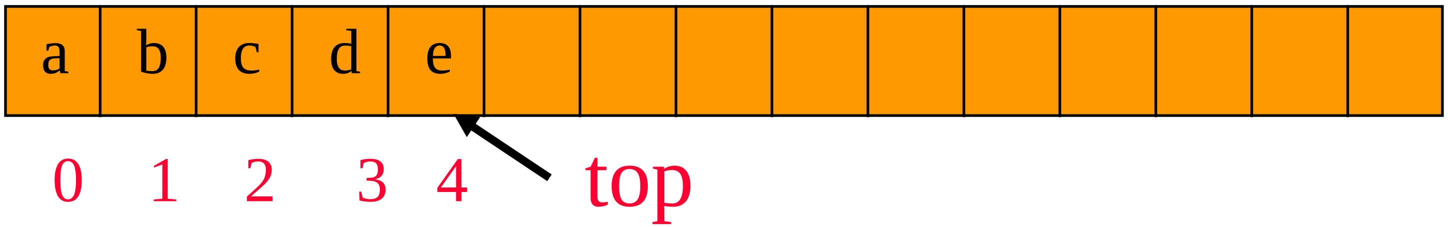
- Use a one-dimensional array stack whose data type is Object.
- Use an int variable size to indicate the number of elements in stack.
  - Stack elements are in stack[0:size-1].
  - Top element is in stack[size-1].
  - Bottom element is in stack[0].
  - Stack is empty iff size = 0.

Array Implementation

```
class ArrayStack:
    def __init__(self):
        self.stack = []
        self.size = 0
```

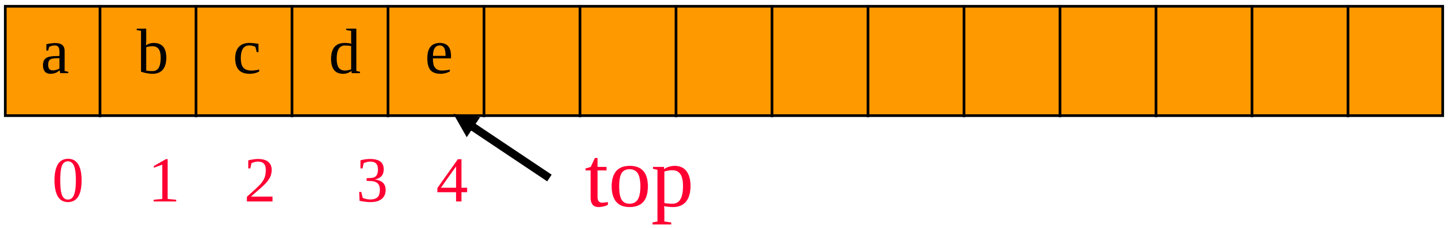
ArrayStack.py

push(…)



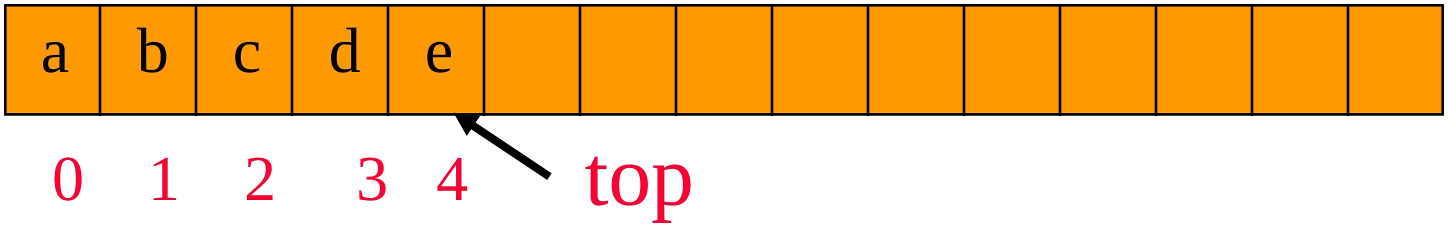

```
#add theElement to the top of the stack
def push(self, element):
      self.stack.append(element)
      self.size += 1
```

peek()



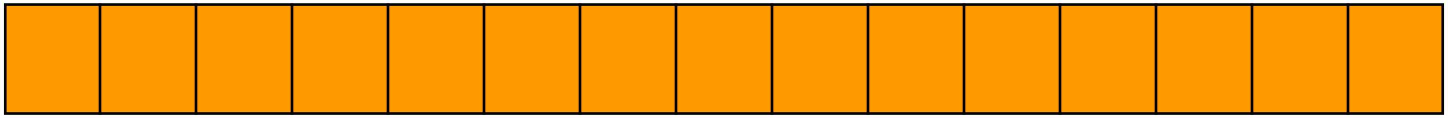

```
# Use peek to look at the top of the stack
def peek(self):
  if(self.size > 0):
    #The last element in the stack
      return self.stack[1]
  else:
      return ("Empty Stack")
```

pop()




```
# Use list pop method to remove element
def pop(self):
    if (self.size > 0):
        self.size -= 1
        return self.stack.pop()
    else:
        return ("Empty Stack")
```

empty()

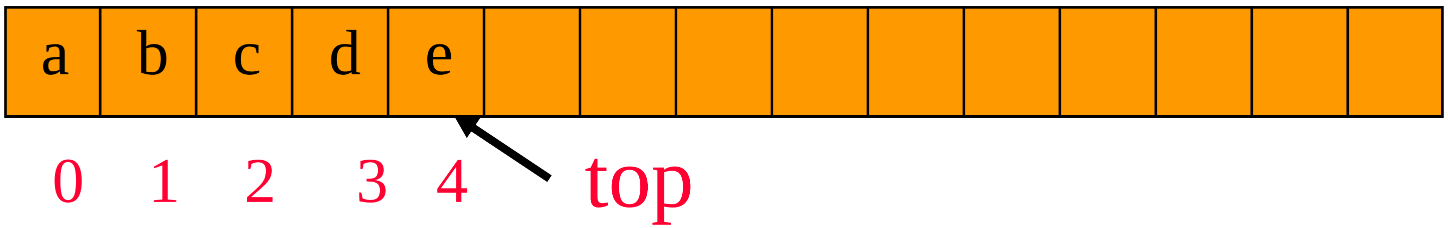


0 1 2 3 4

size = 0

```
#return true if the stack is empty
def empty(self):
    return self.size == 0
```

size()



size = 5

```
#return the size of the stack
def stack_size(self):
     return self.size
```

Applications of Stacks

Applications of Stacks

- Call stack (recursion).
- Searching networks, traversing trees (keeping a track where we are).

Examples:

- Checking balanced expressions
- Recognizing palindromes

  (EYE,or RACECAR, or MADAM I'M ADAM)
- Evaluating algebraic expressions

Simple Applications of the ADT Stack: Checking for Balanced Braces

- A stack can be used to verify whether a program contains balanced braces
  - An example of balanced braces

    `abc{defg{ijk}{l{mn}}op}qr`
  - An example of unbalanced braces

    `abc{def}}{ghij{kl}m`

    `abc{def}{ghij{kl}m`

Checking for Balanced Braces

- Requirements for balanced braces
  - Each time you encounter a “}”, it matches an already encountered “{”
  - When you reach the end of the string, you have matched each “{”

Checking for Balanced Braces

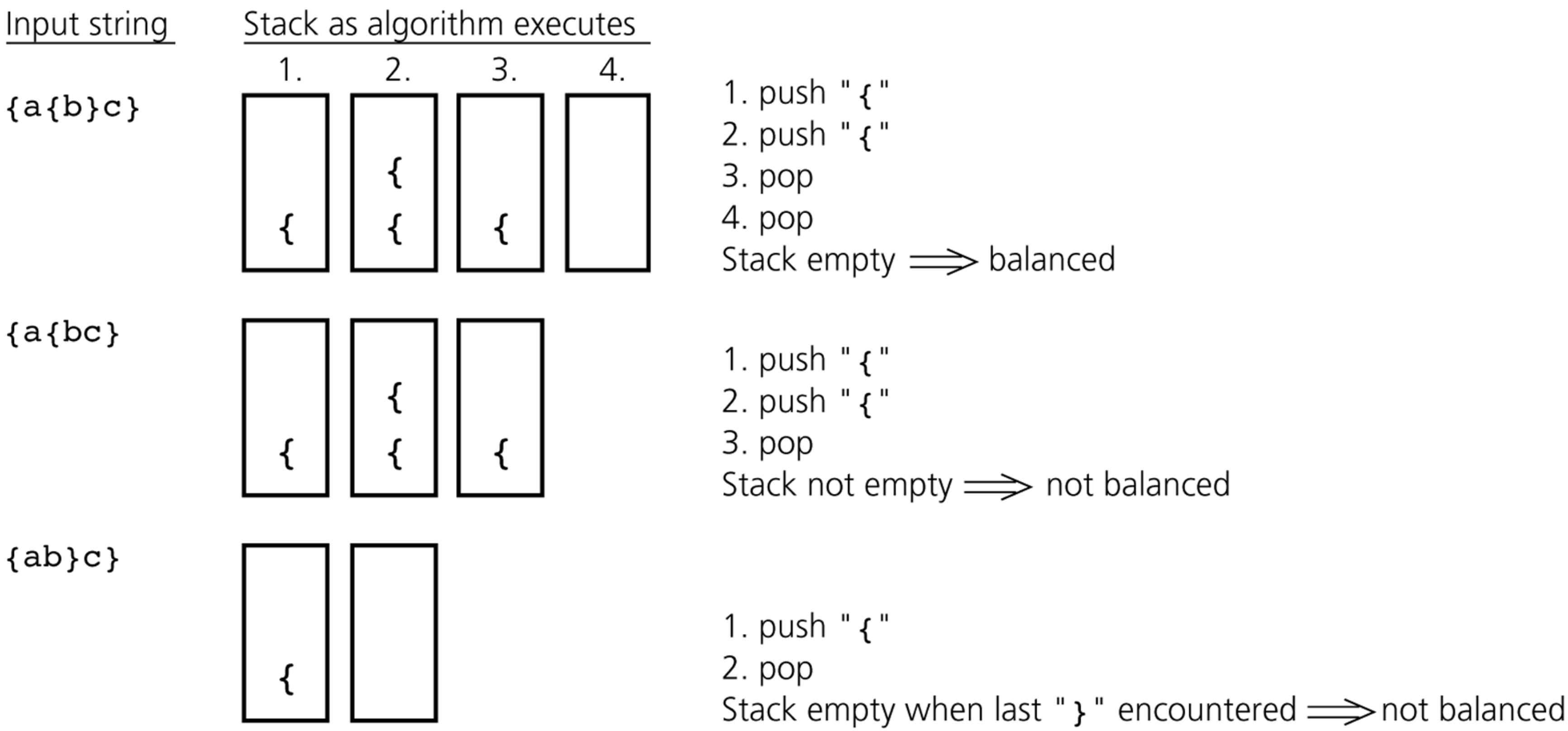


Figure 7-3

Traces of the algorithm that checks for balanced braces

Evaluating Postfix Expressions

- A postfix (reverse Polish logic) calculator
  - Requires you to enter postfix expressions
    - Example: 2 3 4 + *
    - Infix = 2 * (3 + 4)
  - When an operand is entered, the calculator
    - Pushes it onto a stack
  - When an operator is entered, the calculator
    - Applies it to the top two operands of the stack
    - Pops the operands from the stack
    - Pushes the result of the operation on the stack

Evaluating Postfix Expressions

Key entered	Calculator action		Stack (bottom to top)
2	`push 2`		2
3	`push 3`		2 3
4	`push 4`		2 3 4
+	`operand2 = pop stack`	(4)	2 3
	`operand1 = pop stack`	(3)	2
	`result = operand1 + operand2`	(7)	2
	`push result`		2 7
*	`operand2 = pop stack`	(7)	2
	`operand1 = pop stack`	(2)	
	`result = operand1 * operand2`	(14)	
	`push result`		14

Figure 7-8

The action of a postfix calculator when evaluating the expression 2 * (3 + 4)

Evaluate the following Postfix expression

- **3 2 + 4 * 5 1 - /**

Exercise

Queue

Bus Stop Queue

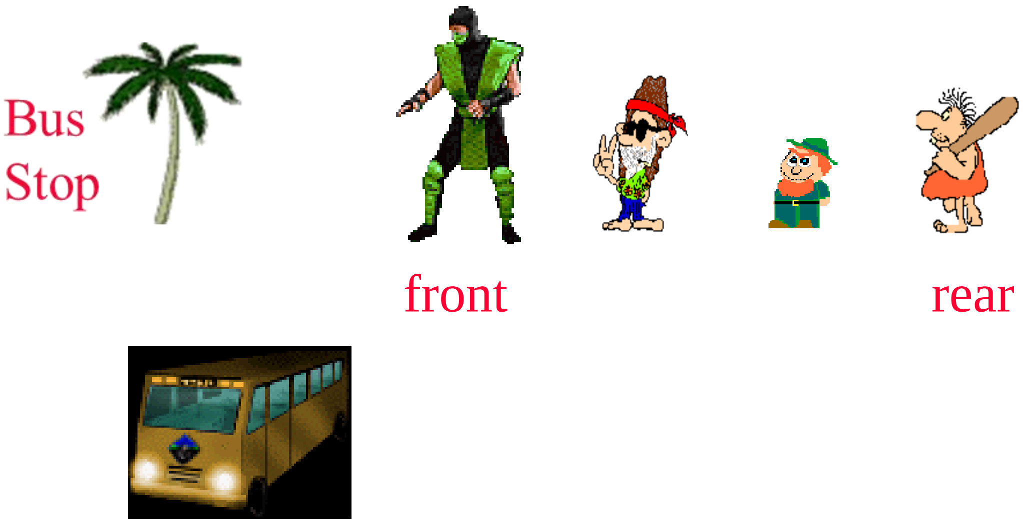
Bus
Stop

front

rear

Add a people to the queue. – **Put / Enqueue**

Bus Stop Queue



Remove a people from the queue. – **Remove / Dequeue**

Bus Stop Queue

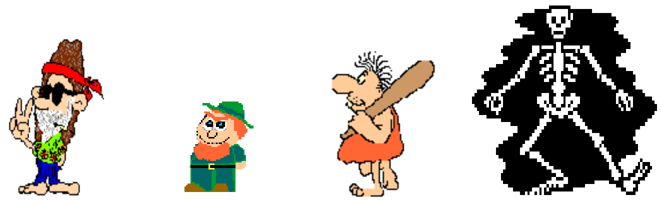
Bus
Stop

front rear

Remove a people from the queue. – **Remove / Dequeue**

Bus Stop Queue

Bus
Stop



front rear

Add a people to the queue. – **Put / Enqueue**

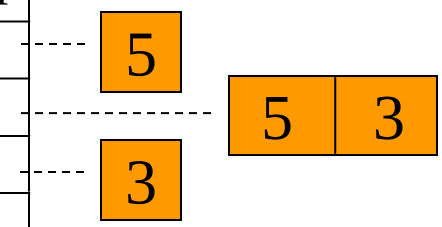
Remove a people from the queue. – **Remove / Dequeue**

A queue is a **FIFO** list. (First In First Out)

Queue operations

This table shows a series of queue operations and their effects. The queue is empty initially.

Operation	Output	front<-Q <- rear
enqueue(5)	-	(5)
enqueue(3)	-	(5,3)
dequeue()	5	(3)
enqueue(7)	-	(3,7)
dequeue()	3	(7)
front()	7	(7)
dequeue()	7	()
dequeue()	"error"	()
isEmpty()	true	()
enqueue(9)	-	(9)
enqueue(7)	-	(9,7)
size()	2	(9,7)
enqueue(3)	-	(9,7,3)
enqueue(5)	-	(9,7,3,5)
dequeue()	9	(7,3,5)



Operation of Queue

isEmpty();

getFrontEelement();

getRearEelement();

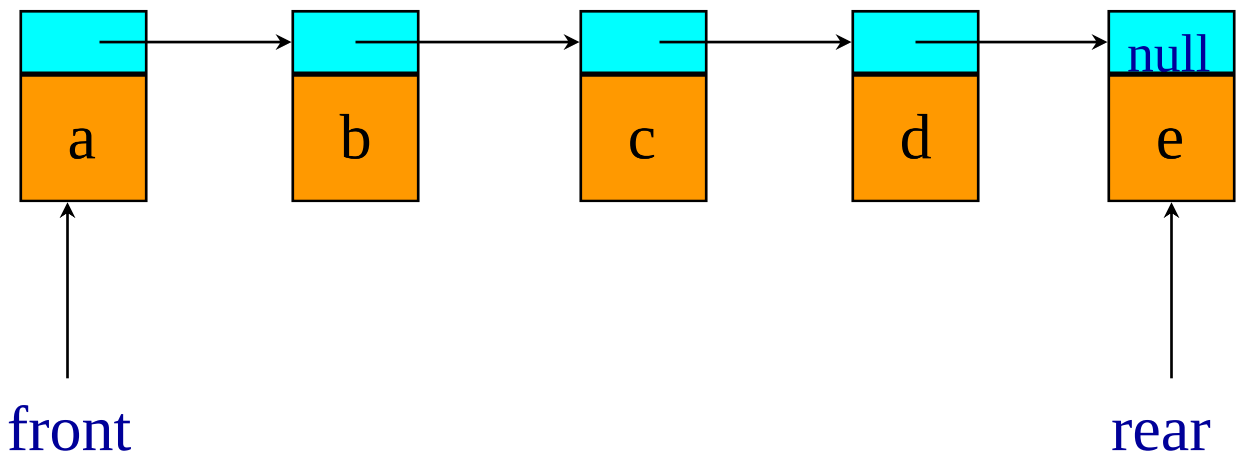
put(Object theObject) **#enqueue**

remove() **#dequeue**

LinkedQueue.py

Linked List Implementation of Queue

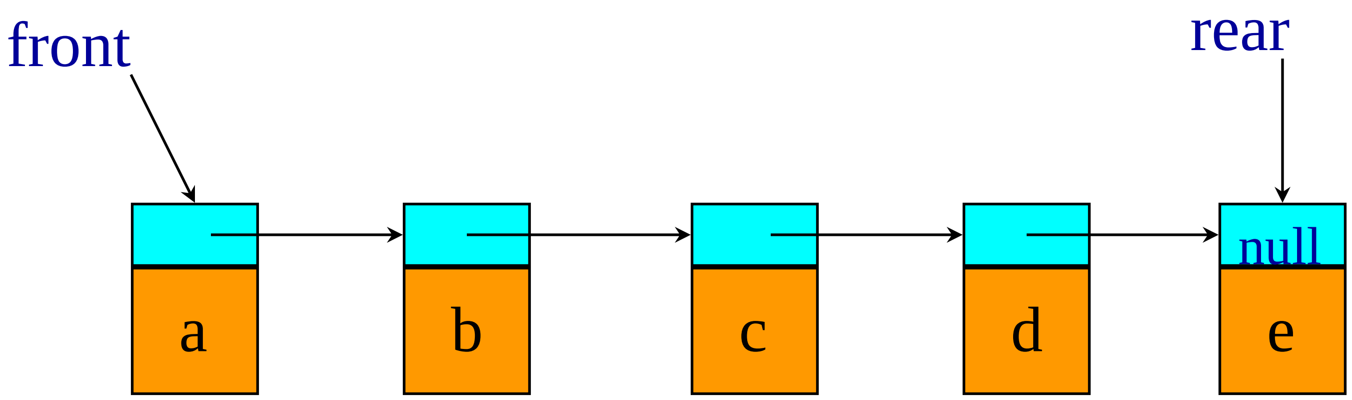
LinkedQueue



- when front is left end of list and rear is right end

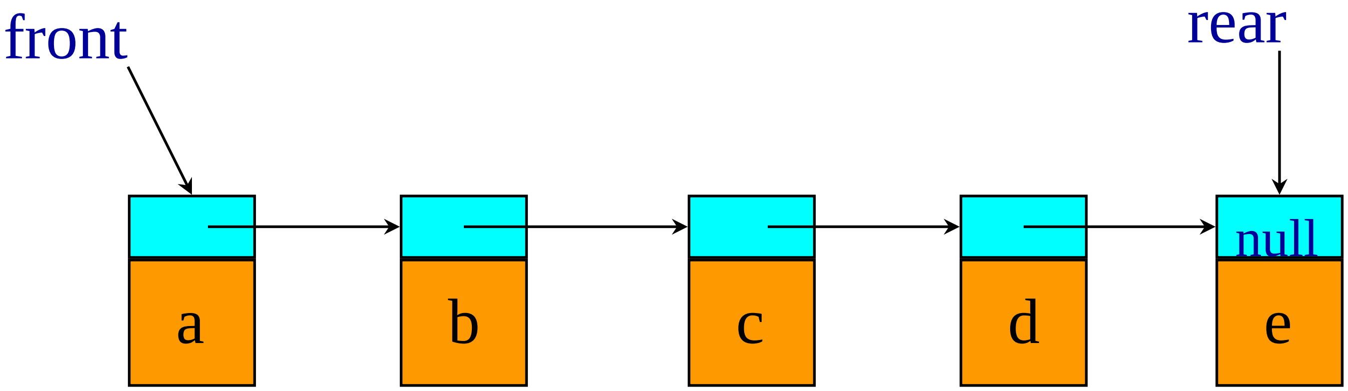
LinkedQueue.py

LinkedQueue



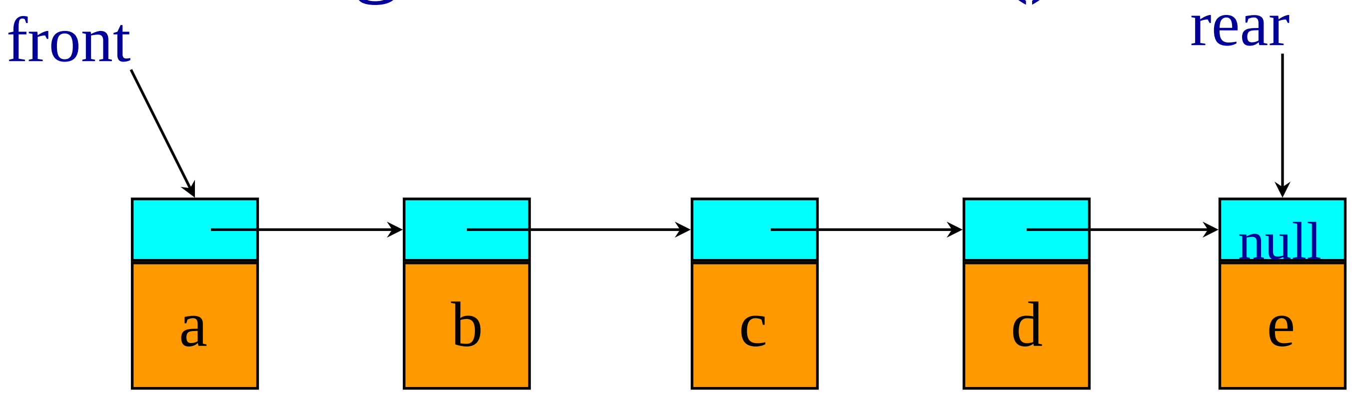

```
class Node:
    def __init__(self, el = None, n = None):
        self.next = n
        self.element = el

class LinkedQueue:
    def __init__(self):
        self.front = None
        self.rear = None
        self.size = 0
```

isEmpty()



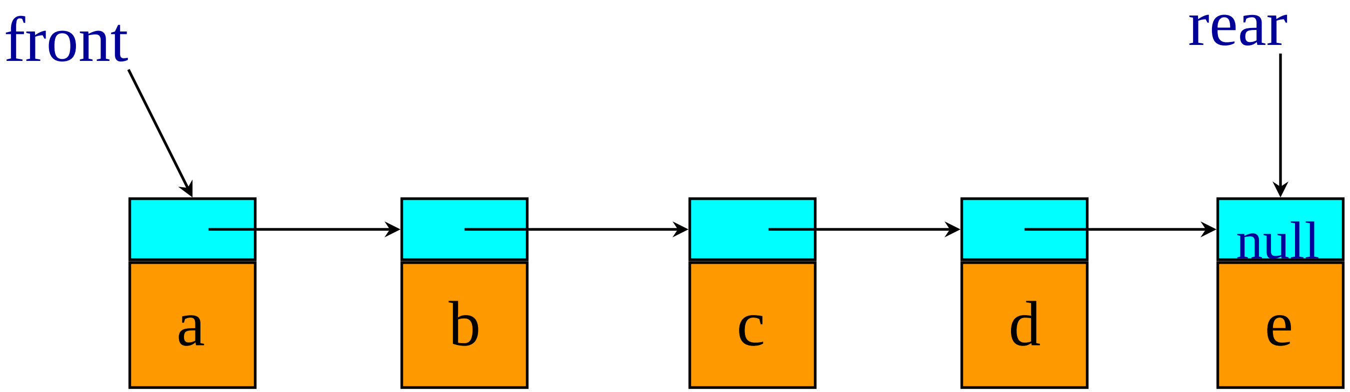

```
#return true if the stack is empty
def isEmpty(self):
    return self.front == None
```

getFrontElement()



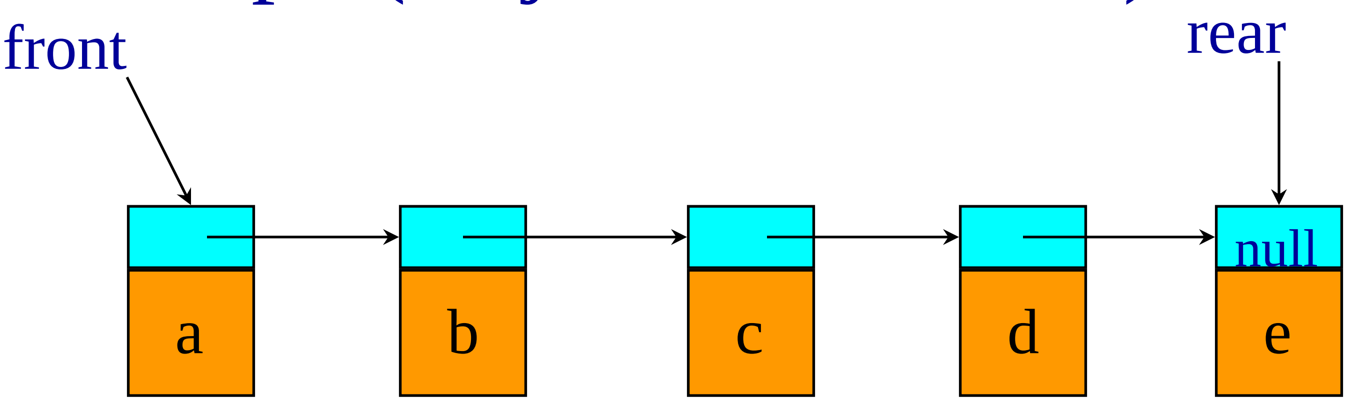

```
#return the first element
def getFrontElement(self):
  if (self.isEmpty()):
      return None;
  else:
      return self.front.element;
```

getRearElement()



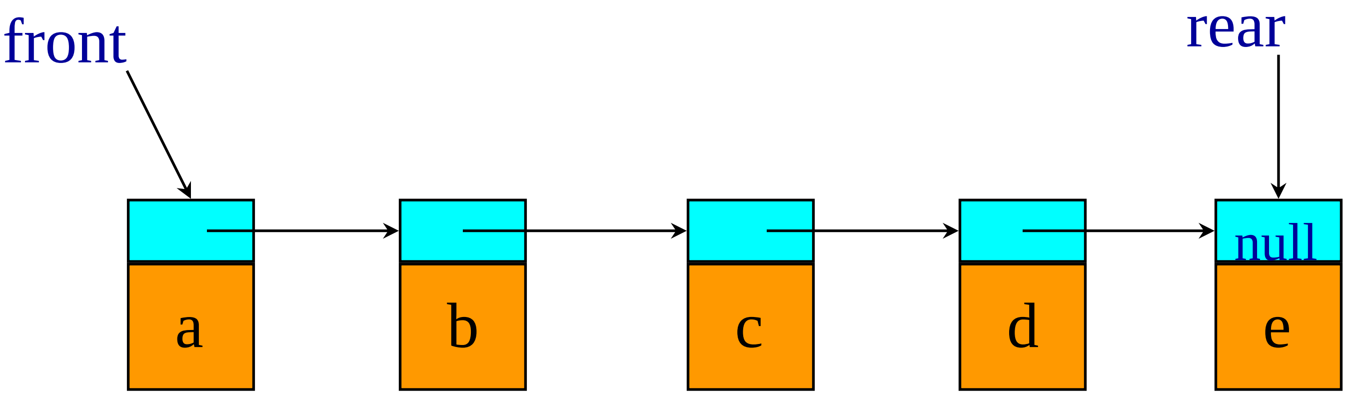

```
#return the last element
def getRearElement(self):
    if (self.isEmpty()):
        return None;
    else:
        return self.rear.element;
```

put(Object theElement)



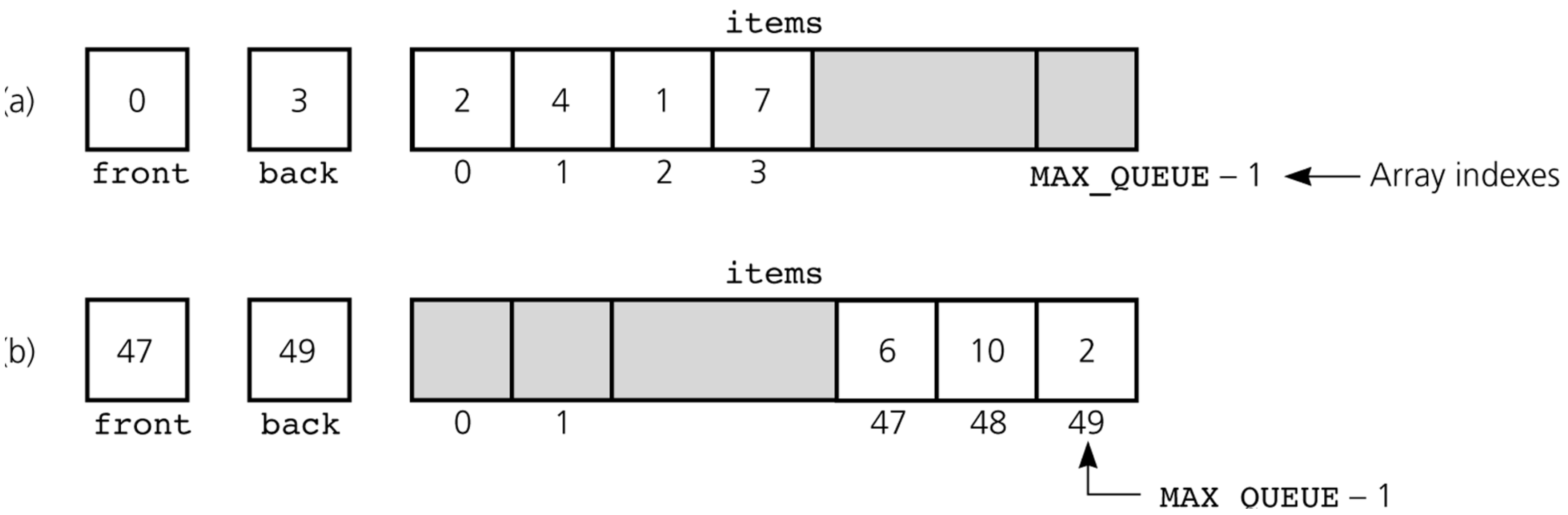

```
#add an element in the queue/enqueue
def put(self, element):
    p = Node(element)
    if(self.isEmpty()):
      self.front = p    #empty queue
    else:
      self.rear.next = p     #nonempty queue
    self.rear = p
```

remove()




```
#remove an element in the queue/dequeue
def remove(self):
    if(self.isEmpty()):
        return None
    frontElement = self.front.element
    self.front = self.front.next
    if(self.isEmpty()):
      self.rear = None
    return frontElement
```

An Array-Based Implementation



a) A naive array-based implementation of a queue; b) rightward drift can cause the queue to appear full

Summary

- Stacks
  - Linear list.
  - One end is called top.
  - Other end is called bottom.
  - Additions to and removals from the top end only.
- Queue
  - Linear list.
  - One end is called front.
  - Other end is called rear.
  - Additions are done at the rear only.
  - Removals are made from the front only.